# A Qualitative Think Aloud Study of the Early Neo-Piagetian Stages of Reasoning in Novice Programmers

Donna Teague and Malcolm Corney
Queensland University of Technology,
Brisbane, QLD, Australia
Tel: +61 7 3138 2000
{d.teague,m.corney}@qut.edu.au

Alireza Ahadi and Raymond Lister
University of Technology, Sydney,
Sydney, NSW, Australia
Tel: +61 2 9514 1850
Raymond.Lister@uts.edu.au

**Abstract**

Recent research indicates that some of the difficulties faced by novice programmers are manifested very early in their learning. In this paper, we present data from think aloud studies that demonstrate the nature of those difficulties. In the think alouds, novices were required to complete short programming tasks which involved either hand executing ("tracing") a short piece of code, or writing a single sentence describing the purpose of the code. We interpret our think aloud data within a neo-Piagetian framework, demonstrating that some novices reason at the sensorimotor and preoperational stages, not at the higher concrete operational stage at which most instruction is implicitly targeted.

*Keywords*: Neo-Piagetian, programming, think aloud.

## 1 Introduction

Recent theoretical and empirical research indicates that the problems faced by many novice programmers start very early. In empirical work, Corney, Lister and Teague (2011) collected data showing that students who fared poorly on tests held as early as week 3 of semester were less likely to perform well on a code writing task at the end of semester. That empirical result has since been replicated at other institutions (Corney, Teague, Ahadi, and Lister, 2012; Murphy, McCauley, and Fitzgerald, 2012). In theoretical work, Robins (2010) has produced a statistical model to explain the commonly observed bimodal grade distribution, based on the principle that if a student struggles at an earlier point in semester, then the student is more likely to struggle later in semester.

One of the questions used by Corney, Lister and Teague (2011) in their empirical work is shown in Figure 1. They gave this question to students in their fifth week of learning to program in Python. Less than half of their students answered this question correctly. The literature on novice programmers contains three perspectives as to why students might struggle to answer that question, but there are also arguments against each of those perspectives:

- *Programming Misconceptions*: There have been many studies of novice misconceptions (e.g. Du Boulay, 1989). However, Corney, Lister and Teague (2011) used a pre-test to screen out students who had misconceptions about variables, assignment statements and `if` statements.
- *Misunderstanding what was required*: Perhaps the students thought they were required to provide a line-by-line description of the code. Corney, Lister and Teague (2011) argued that this was implausible for three reasons. First, the question was constructed to indicate what type of answer was required. Second, their students had already encountered an "explain in English" question in an earlier test, and had been shown an appropriate sample answer for that earlier question. Third, Corney, Lister and Teague (2011) indicated that most incorrect answers were of the right type, but were simply wrong (e.g. "swap y1 and y3").
- *Poor Self Expression in English*: Perhaps the students knew the correct answer, but could not express that answer? To investigate that possibility, Simon and Snowdon (2011) gave multiple choice versions of code explanation questions to their students, so that students merely had to select the correct answer, rather than express it for themselves. Simon and Snowdon (2011) found that a non-trivial portion of their students selected the wrong option, so poor self expression in English was not the problem for those students.

A more comprehensive refutation of the above three perspectives requires direct observational evidence identifying other reasons why students struggle. In this paper, we provide such evidence, via a think aloud study. It is a small study, involving only seven subjects, but that proved sufficient to identify other reasons why students struggle. Our aim was not quantitative. That is, our aim was not to establish how common those other reasons are in the general population of students.

Observations contrary to the perspectives in the above three bullet points would be even more persuasive if accompanied by an explanatory theory. Lister's neo-Piagetian framework (2011) provides such a theory. In the next section we describe that framework, before presenting our observations from the think aloud study, in terms of that neo-Piagetian framework.

If you were asked to describe the purpose of the code below, a good answer would be "*It prints the smaller of the two values stored in the variables* `a` *and* `b`".

```
if (a < b):
    print a
else:
    print b
```

**In one sentence that you should write in the empty box below, describe the purpose of the following code.**

Do **NOT** give a line-by-line description of what the code does. Instead, tell us the purpose of the code, like the purpose given for the code in the above example (i.e. "*It prints the smaller of the two values stored in the variables* `a` *and* `b`").

Assume that the variables y1, y2 and y3 are all variables with integer values.

In each of the three boxes that contain sentences beginning with "`Code to swap the values …`", assume that appropriate code is provided instead of the box – do **NOT** write that code.

```
if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
if (y2 < y3):
    Code to swap the values in y2
    and y3 goes here.
if (y1 < y2):
    Code to swap the values in y1
    and y2 goes here.
print y1
print y2
print y3
```

*Sample answer*:

It sorts the values so that y1 ≥ y2 ≥ y3

**Figure 1: A question from the week 5 test of Corney, Lister and Teague (2011).**

## 2 The Neo-Piagetian Stages

Classical Piagetian theory focuses on the general intellectual development of children as they mature. Neo-Piagetian theory instead describes the intellectual development through which people progress, regardless of age, as they gain expertise in a specific problem domain, such as chess, or programming.

In the context of neo-Piagetian theory, Lister (2011) described four main stages of cognitive development in the novice programmer, which are (from least mature to most mature) sensorimotor, preoperational, concrete operational and formal operational. The question in Figure 1 is not designed to elicit formal operational reasoning from novices, so that stage is not discussed any further in this paper. The other three stages are described below, from least mature to most mature.

### 2.1 The Sensorimotor Stage

Lister (2011) describes a programmer operating at the sensorimotor stage as a novice who is unable to accurately and reliably hand execute ("trace") small pieces of code to determine the final values in the variables.

There are two broad reasons why some novices do not reliably trace code. One reason is that some novices have misconceptions, which is well documented in the literature (e.g. Du Boulay, 1989).

Another reason is that for some novices the effort of tracing is so great, and the likelihood of a correct trace so small, those novices simply do not wish to trace. In their multinational study, Lister et al. (2004) found that many students who were asked to trace code on a piece of paper returned that paper unmarked. Even among students who do trace, those at the sensorimotor stage find the low level mechanics of conducting a trace so demanding that it does not help them to "see the forest for the trees". A sensorimotor novice is reluctant to trace a piece of code more than once. If required to generate their own initial data for a trace, the sensorimotor novice tends not to choose data that will help them better understand the code.

Later in this paper, we will present think aloud data for Donald, who demonstrates his understanding of variables, assignment statements and `if` statements, and also demonstrates a willingness to trace, but struggles to organise his tracing in a way that produces reliable results.

### 2.2 The Preoperational Stage

Preoperational is the next stage after sensorimotor. Novices operating at this stage can trace code accurately and efficiently.

When asked to answer the question in Figure 1, a novice working at the preoperational stage uses an inductive approach. That is, such a novice may perform one or more traces and then make an educated guess based on the input/output behaviour. Later in this paper, we will present think aloud data collected from Lucas and Sierra, who manifest such preoperational reasoning.

Kolikant and Mussai (2008) described how, when given buggy programs to comment upon, some novice programmers viewed the programs as partially correct. This notion of partial correctness is consistent with novices working at the preoperational stage.

### 2.3 The Concrete Operational Stage

A novice reasoning at the concrete operational stage is capable of deductive reasoning. That is, a novice reasoning at the concrete operational stage should answer the question in Figure 1 quickly and correctly, simply by reading the code. They should not need to perform an explicit, complete written trace to arrive at the answer.

Instead of reasoning in terms of specific values within variables, as the preoperational novice does, the concrete operational novice reasons about code in terms of

constraints on the possible values of variables. For example, after the body of the second `if` statement in Figure 1, the concrete operational novice thinks of y3 as holding any value satisfying the condition that it is less than the values in both y1 and y2. Furthermore, by the time the code in Figure 1 has completed execution, the concrete operational novice infers that because y1 is greater than y2, and y2 is greater than y3 then y1 is greater than y3 — that is, the variable values are in descending order. In neo-Piagetian theory, such reasoning is known as transitive inference.

## 3 Method

### 3.1 Think Aloud Sessions

We conducted one-on-one think aloud sessions with volunteer students from the first two introductory programming units at QUT during the second half of 2011. The students were asked to complete a series of simple programming tasks while thinking aloud. Ericsson and Simon (1993) developed protocols for eliciting simple unstructured verbalisations. They emphasised the need to minimise the cognitive effort in producing such verbalisations. The goal is to have the subject simply articulate what is going on in their head, rather than formulate an explanation or description for the benefit of the interviewer. Only when the subject is completely focussed on the programming task can we expect to replicate a silent attempt at the same task with the same or similar sequence of thoughts by the subject. Prior to the data collection reported in this paper, all subjects were given the opportunity to practice thinking aloud and to become at ease with the interviewer and familiar with the technology used to record the sessions.

Each think aloud session was recorded using a Smartpen (2011) which digitally captures whatever is written on special dot paper. The digital "pencast" PDFs produced by the Smartpen can be replayed with synchronised visual and audio output using Adobe Acrobat Reader. The sessions can then be replayed during analysis and shared without the need for special technology.

The students participated in think aloud sessions on a more-or-less weekly basis over a semester, each session lasting for roughly 60 minutes.

### 3.2 The Subjects

IT students at most universities progress through one or more programming units which we will refer to by level. Level 1 refers to the first introductory programming unit with zero pre-requisites. Level 2 units have Level 1 units as a pre-requisite. It is the performance of students at these first two levels that we will discuss in this paper.

To preserve their anonymity, each of the seven students who took part in think aloud sessions chose an alias. Some details about these students at the time of participating in this study are shown in Table 1. Information in that table includes the current programming level at which each student was studying, their Level 1 result, and the week in semester when the think aloud session occurred.

| Alias | Level | Level 1 result | Week |
|---|---|---|---|
| Stapler | 2 | top 21% of cohort | 10 |
| Becki | n/a | top 21% of cohort | 11 |
| Donald | 2 | top 69% of cohort | 9 |
| John | 1 | top 17% of cohort | 6 |
| Mel | 1 | top 43% of cohort | 6 |
| Lucas | 1 | top 43% of cohort | 6 |
| Sierra | 1 | top 43% of cohort | 6 |

**Table 1: The seven student subjects**

Two of the students, Lucas and Sierra, chose to participate in think aloud sessions as a pair, as they were already a pair in their Level 1 unit programming labs.

### 3.3 The Tasks

Prior to our students completing the task from Corney, Lister and Teague (2011) shown in Figure 1, they each completed tasks which tested their ability to trace code, and explain in plain English the purpose of given code. According to neo-Piagetian stage theory in the programming domain, the inability to trace reliably (or at least to have great difficulty with tracing) is indicative of a novice operating at the sensorimotor stage. The inability to explain code indicates a stage no higher than preoperational.

As Stapler, Becki and Donald were "post Level 1" students, the tasks they were given were a mix of Python (the Level 1 language) and C# which is the language introduced in Level 2. (Even though Becki had no previous exposure to C#, she was confident that she would be able to interpret C# code adequately enough to answer the questions.)

The tracing task given to the Level 1 students is shown in Figure 2. The other students were given the tracing task shown in Figure 3.

To test their ability to reason about code and provide an explanation in plain English, two tasks were used. Level 1 students were given code that swapped the values in two variables using a third as temporary storage as shown in Figure 4. This problem was first used by Corney, Lister and Teague (2011).

Level 2 students (and Becki, who had completed Level 1 but was not doing the Level 2 subject) had to explain more complicated code than just simple assignments. Their code included nested conditional blocks which found the middle of three values. The task not only tested the ability to extract a meaning from the given code, but also tested a student's ability to reason by transitive inference. This task is shown in Figure 5.

Write the values in the variables after the following code has been executed:

```
x = 7
y = 5
z = 3
x = y
z = x
y = z
```

**Figure 2: Level 1 tracing task**

Write the values of the specified variables after all of the statements have been executed:

```
a = 7
b = 3
c = 2
d = 4
e = a
a = b
b = e
e = c
c = d
```

**Figure 3: Level 2 tracing task**

The purpose of the following three lines of code is to swap the values in variables a and b, for any set of possible initial integer values stored in those variables:

```
c = a
a = b
b = c
```

In one sentence that you should write in the box below, describe the purpose of the following three lines of code, for any set of possible initial integer values stored in those variables:

```
j = i
i = k
k = j
```

Sample answer:

*It swaps the values in variables i and k*

**Figure 4: Level 1 Explain in Plain English task**

## 4 Results

### 4.1 Performance on the Tracing Tasks

John and Mel had no difficulty with the tracing task in Figure 2. They each accurately verbalised and wrote down the changing value of the variable at each assignment. Similarly, Stapler and Becki were both able to perform an accurate trace of the code in Figure 3 without any difficulty. This provides evidence that these four students were operating *at least* at the preoperational stage.

Although Lucas and Sierra eventually traced the code in Figure 2 accurately, there were some initial indications of a misconception about variable assignment. For example, at one point Lucas read the line "y = z", but articulated "y is assigned to z". This could be interpreted as a clear misunderstanding of the direction of assignment, or alternatively as evidence of cognitive overload causing confusion and an inability to accurately articulate their understanding. After some discussion with his partner Sierra, they agreed on the direction of the assignment, and established a somewhat clearer method of articulating this. The greater cognitive effort required by this pair to correctly trace the code indicates that for them the concepts of variables and assignment were more fragile than for John, Mel, Stapler and Becki. However, their ability to complete the tracing task provides evidence that they may be operating at the preoperational stage.

There are two initialised integer variables in scope: **`first`** and **`second`**. If you were asked to describe the purpose of the code below which uses those variables, a good answer would be "*It outputs the highest value.* "

```
if (first >= second) {
    Console.WriteLine(first);
} else {
    Console.WriteLine(second);
}
```

In one sentence that you should write in the empty box below, describe the purpose of the following code, where `first`, `second` and `third` are integer variables which are all initialised and in scope:

```
int maximum = Math.Max(first, second);
maximum = Math.Max(maximum, third);
int minimum = Math.Min(first, second);
minimum = Math.Min(minimum, third);

if (first == minimum) {
       if (second == maximum) {
              Console.WriteLine(third);
       } else {
              Console.WriteLine(second);
       }
} else if (second == minimum) {
       if (first == maximum) {
              Console.WriteLine(third);
       } else {
              Console.WriteLine(first);
       }
} else {
       if (first == maximum) {
              Console.WriteLine(second);
       } else {
              Console.WriteLine(first);
       }

}
```

Sample answer:

*Finds the middle integer value*

**Figure 5: Level 2 Explain in Plain English task**

Donald, on the other hand, experienced great difficulty with his tracing task. He demonstrated very poor tracing skill. He lost his way many times over the considerable period of time it took him to complete the task. Donald was unable to keep track of the changing values in the variables and exhibited increasing frustration during his several attempts to complete the trace.

Table 2 summarises our classification of the seven students, on the basis of their performance on these

tracing tasks. Tracing code is a task performed reliably and accurately by a preoperational student, so John, Mel, Stapler and Becki provided evidence for being *at least* at the preoperational stage. The performance of Lucas and Sierra is not so easy to classify, but after much discussion between them they did eventually successfully complete their tracing task, so perhaps they are at the preoperational stage. Donald, on the other hand, showed little capacity to trace. He is probably at the sensorimotor stage.

### 4.2 Performance on the Explanation Tasks

For those students who we classified as being *at least* preoperational on the basis of the tracing task, a further test is required to determine if they are actually at the concrete stage, which is the next neo-Piagetian stage. Our further testing entailed observing their ability to extract meaning from code — to "explain in plain English".

John completed the explaining code task shown in Figure 4. He started by writing down the three lines of code that are given in the question and then, after thinking for a short period of time, determined that the code swapped the values in `i` and `k`. We classified his performance as concrete operational.

Mel simply read the question and established that this code was "also swapping". As Mel did not elaborate on which variables' values were being swapped, we cannot be absolutely confident that she had established that `i` and `k` were swapped. She could have meant that the code changed the values of all the variables, although the use of the word "swapping" might indicate a more specific function, albeit ambiguous. We made a determination of Mel's ability to reason about code as being concrete operational, based upon her performance on the problem discussed in Section 4.3 and also on other problems Mel has done, which are not described in this paper.

Stapler carefully read the question in Figure 5 and wrote down the purpose of `maximum` and `minimum` variables. After reading only a small portion of the remaining code, he drew a quick and accurate conclusion about its purpose. Becki also completed this task quickly.

In solving this problem, our four *at least* preoperational stage students all provided evidence suggesting they are operating at the concrete operational stage. They could see the relationships between different elements of a piece of code and were able to explain the overall purpose of that code. On the basis of this evidence, and the evidence from the earlier tracing task, we classify our seven students as operating at the neo-Piagetian stages shown in Table 3.

### 4.3 Performance on the Question in Figure 1

We now discuss the performance of all seven of our students on the question in Figure 1. The four students listed in Table 3 as being concrete operational also manifested concrete operational reasoning on this problem. Three of those four novices provided a completely correct answer. The exception was John, who nominated ascending order rather than the correct descending order. We regard that as a minor oversight by John.

None of the four students at the concrete operational stage completed a written trace. All but one of them wrote nothing except their answer. The exception was Becki, who wrote down "`4 < 3`" before crossing it out a few seconds later, and then she wrote nothing else except her final answer.

| Alias | Neo-Piagetian Stage |
|---|---|
| Stapler | *at least* preoperational |
| John | *at least* preoperational |
| Becki | *at least* preoperational |
| Mel | *at least* preoperational |
| Lucas & Sierra | preoperational |
| Donald | sensorimotor |

**Table 2: The manifested neo-Piagetian stages of the seven students on their tracing tasks**

| Alias | Neo-Piagetian Stage |
|---|---|
| Stapler | concrete operational |
| John | concrete operational |
| Becki | concrete operational |
| Mel | concrete operational |
| Lucas & Sierra | preoperational |
| Donald | sensorimotor |

**Table 3: The manifested neo-Piagetian stages of the seven students after their Explanation Task**

| | Students | | | |
|---|---|---|---|---|
| | **Stapler** | **John** | **Becki** | **Mel** |
| **Seconds to read preamble** | 59 | 54 | 31 | 70 |
| **Seconds to read code and write answer** | 78 | 179 | 69 | 115 |
| **Total Time** | 2 mins 17 secs | 3 mins 53 secs | 2 mins 10 secs | 3 mins 5 secs |

**Table 4: The time taken by the concrete operational students to answer the question in Figure 1.**

Table 4 summarises the time taken by these four concrete operational students to answer the question. The row in that table "Seconds to read preamble" shows the amount of time that elapsed from when each student started reading the question until the student reached "do NOT write that code". The next row in the table shows the remaining time each student spent on the task.

John took longer than the others to answer the question, because he was briefly puzzled by the purpose

of the third `if` statement. While considering that statement, he uttered "*Why would you repeat that line of code*?" In asking that question, he explicitly manifested concrete operational reasoning.

With the exception of John, the data presented here does not prove that these students were reasoning at the concrete operational stage. It is possible that a novice operating at the preoperational stage might perform a single trace of this code in their head, and then make a correct inductive guess. However, based on the earlier think aloud data we presented for these four students, it is likely these students are also reasoning at the concrete operational stage on this problem. Our intent in presenting this data for these students is not to provide conclusive evidence that they reasoned at the concrete operational stage, but instead to provide a point of reference, a contrast, to the three novices for whom we next present think aloud data − Donald, Lucas and Sierra.

## 5 Pre-Concrete Operational Students

### 5.1 Donald − Sensorimotor

The think aloud session described here was Donald's fifth such one-hour session, so he was familiar with the process by this time.

In his first 60 seconds, Donald read the question aloud, from the beginning down to "do NOT write that code" (i.e. just above the code he needed to describe). While reading aloud, he manifested good English reading and language skills.

He next read the code, for about 35 seconds, without writing anything down. As he read, he articulated the following description of the code:

> *So … if y1 is less than y2, code to swap the values in y1 and y2 goes here … so if it's less than … then the greater number goes into y1, if y2 less than y3 … goes here … here. Okay.*

(Note that Donald says "*so if it's less than … then the greater number goes into y1*". In the subsequent effort to complete the trace, that crucial observation is later forgotten by Donald.)

Donald then began the annotations shown in Figure 6. (The annotations "Line 1", "Line2" and "Line 3" are not Donald's work, but were added by the authors of this paper.) After beginning by writing "y1" on Line 1, Donald said "*hypothetically let's say y1 was 1*". He then wrote that number above "y1". After writing out the rest of the code on Line 1, he wrote a "2" above "y2". He then added the arrow above Line 1, while saying "*So y1 would get number 2*". From writing the initial "y1" to drawing the arrow, there is little hesitation, and only about 15 seconds elapse.

Donald then began producing line 2 in Figure 6. He first wrote "y2 < y3" followed by a "2" above "y2". That "2" is obscured in the figure by a correction made later (see below), as "2" is not the correct value for y2 after the execution of Line 1. Donald then wrote a "3" above "y3" and drew the arc from "y3" to "y2". He then sporadically muttered variable names for 15 seconds, before saying "*Wait … what have I done*?" After a 10 second pause he said "*Oh yeah*", and corrected his earlier error by writing a "1" over the top of the "2". He then correctly updated his annotations to show the values of y2 and y3 being swapped, by writing a "3" above and to the left of "y2" and "1" above and toward the right of "y3". However, he did not cross out the previous values in those two variables, and that may be the source of his subsequent confusion (as will be described in the next paragraph). He said at this point, "*Probably should write that out clearer, but anyway …*" From his first annotation on line 2 to his last annotation on that line, 57 seconds elapse, which is much longer than his time to complete Line 1.

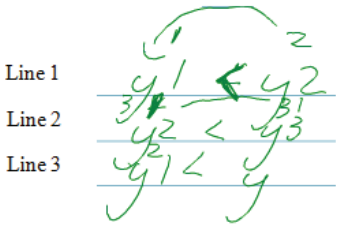


**Figure 6: Donald's first and unsuccessful trace.**

Donald then began his third line of annotations. He first wrote "y1" and immediately added "2" above it, while uttering "*y1 was now holding number 2*". He then wrote the remainder of the third line, while uttering "*and y2 is holding …*" after a pause of several seconds he commented "*it's a poor way to write it out for myself*". From his first annotation on line 3 to his last annotation on that line, 20 seconds elapse.

Even though Donald wasn't able to complete his first trace, the effort of attempting that trace did then lead him to articulate a tentative idea as to what the code does, but it is a wrong idea:

> *"I think what it's doing is just swapping them all down, so reversing. So, it swaps, ultimately what's in y1 with y … 3? … Oh my God, if I could do this clearer it would be much easier to figure out."*

Donald then began a second and clearer trace, by writing the annotations shown in Figure 7. He first wrote, on each of the three lines, respectively "y1 = 1", "y2 = 2" and "y3 = 3". He then performed a conventional and correct trace, which took 67 seconds.

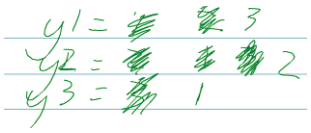

**Figure 7: Donald's second trace of the code, which was his first successful trace.**

Despite having just completed the correct trace shown in Figure 7, he then re-articulates the same wrong idea about what the code does:

> *"So, what we've done is reverse the variables … swap … So, let's say that in a nice elegant sentence: It swaps the variables, the order of the*

*variables ... no, it reverses the order of the variables? Yeah?"*

After a few more seconds of broken muttering, he writes his incorrect answer:

*"To reverse the values stored in y1, y2 and y3 and then print them to screen."*

At this point, almost six minutes have elapsed since Donald first began reading the question, and almost four minutes have elapsed since he began to read and trace the code, which is slower than the times shown in Table 4 for the concrete operational students.

After Donald indicated to the interviewer that he had finished the question, the interviewer asked Donald to trace the code again, using the initial values y1 = 2, y2 = 1 and y3 = 3. After Donald had written down those three initial values, as shown in Figure 8, he proceeded to perform a correct trace, taking about 70 seconds to do so, including a short pause (perhaps a pause of surprise) when the first `if` block did not cause the values in y1 and y2 to be swapped. On completing the trace, however, Donald initially maintained that the code "*ended up the same ... as what I originally came up with*". (His tone of voice, however, may suggest this utterance was more a question to the interviewer than a committed statement, or perhaps even an ironic remark. In any event, he certainly does not articulate at this point an alternative description of the code.) After being challenged on the correctness of that assertion by the interviewer, but without the interviewer hinting any further as to what the correct answer might be, Donald exclaimed:

*"Oh! It's ordering them ... um ... so, it's more about, it's not to rev ... hang on ... oh* [indecipherable]*... rather than to reverse, it would be to, place them from highest to lowest."*

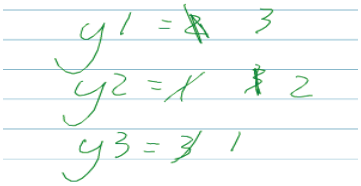

**Figure 8: Donald's third trace of the code, using initial values provided by the interviewer.**

## 5.2 Interpretation of Donald's Performance

While Donald did successfully trace the code at his second attempt, his first attempt demonstrates that he struggles to organise his tracing in a way that efficiently produces reliable results. Despite successfully tracing the code at his second attempt and thus having read the code closely, his subsequent answer is really a guess. Even if he had performed a third trace spontaneously, it is not clear whether Donald could have specified for himself suitable initial values that would have tested his guess. Because tracing is a difficult process for Donald, he only tests his guess with a third trace when asked to do so by the interviewer. It also falls to the interviewer to provide initial values that will test Donald's guess.

When Donald does abstract from the code, his abstractions are localised, so that an abstraction he makes at one moment can be inconsistent with a subsequent abstraction. For example, early in the process of reading the code, Donald says "*so if it's* [viz. y1] *less than* [y2]*... then the greater number goes into y1*", but that abstraction is forgotten by, and inconsistent with, his subsequent answer, "*reverse the values stored in y1, y2 and y3*".

## 5.3 Lucas / Sierra − Preoperational

Lucas and Sierra elected to work together in think aloud sessions because they were already working as a pair in weekly programming laboratory sessions. Their approach to think aloud sessions reflected their pair-programming laboratory sessions – one of them would write and describe what he was doing, while the other monitored and intervened when he felt it was necessary. This was their third think aloud session, so they were well practiced. In this session, and also in earlier sessions, Lucas and Sierra manifested good language skills.

In their first 60 seconds, Sierra read the question aloud from the beginning down to "do NOT write that code" (i.e., just above the code they needed to describe). They then read and discussed the code, for about 50 seconds, without writing anything down. The following is a transcript of that discussion:

Sierra: *If y1 is less than y2 ...*

Lucas: *... code to swap the values in y1 and y2 goes here, yeah.*

Sierra: *If y2 is less than y3, swap the two.*

Lucas: *Yep. If y1 is smaller than y2, okay, so if that is smaller than that, their values swap, if that is smaller than that, after that, their values swap, and if that is smaller than that, then their values swap; does that mean they just ... changing the order of the ascending and the descending? Like, for instance, I'm just gonna write it out.*

Lucas then wrote down the first line shown in Figure 9. As he did that, Sierra suggested the initial values shown in the boxes on that line. Next, they collaborated on a smooth and successful trace, which took 42 seconds.

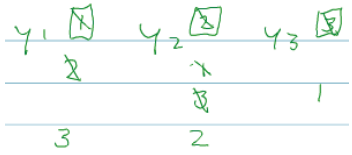

**Figure 9: The first trace by Lucas and Sierra.**

The following discussion then ensued:

Lucas: *Thus, we swap the order ...*

Sierra: *But we don't want to talk about each line of code. Basically, we are just changing the order of code from …*

Lucas: *We're changing the integer values, swapping it, but what if … let's do a "what if" scenario …*

Sierra: *It's changing it from, basically ascending order to descending order.*

Lucas: [While writing the first line shown in Figure 10.] *If these are the correct integers. I'm just going to try it the other way around, so make y1 three, make y2 two and y1 one.*

They then collaborated on completing another smooth and successful trace, as shown in Figure 10.

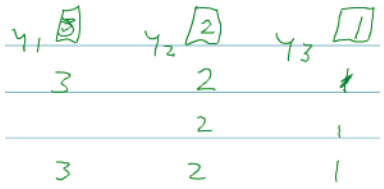


**Figure 10: The second trace by Lucas and Sierra.**

As Lucas wrote the second last line of Figure 10, the conversation continued:

Sierra: *I see what they are doing now. It's like what I said before …*

Lucas: [While writing the final line in Figure 10] *… and if y..., that's still the same, so it just changes it, from ascending, ah, yeah, from largest to smallest … it will consistently be that.*

Sierra: *That's what we've got to write.*

Lucas then commences writing their answer while saying it aloud:

Lucas: *"The purpose of the code is to …*

Sierra: *… change the order? From …*

Lucas: *Well, not necessarily change it, because if it is already in that order, then it will … not change it, what's another word?*

Sierra: *Is to, umm, rearrange?*

Lucas: *Yeah, I guess … to arrange, not rearrange, because that implies again that you are moving the integers.*

Sierra: *Alright.*

Lucas then completes writing their final, correct answer:

"*… arrange variables from highest to lowest.*"

At this point, almost 5 minutes have elapsed since Lucas and Sierra began reading the question, and almost 4 minutes have elapsed since they began to read and trace the code, which is longer than the times for the concrete operational students in Table 4.

### 5.4 Interpretation of Lucas and Sierra's Performance

Lucas and Sierra are clearly more advanced novices than Donald, as they traced the code faster and accurately. However, they are not as advanced as the four students who manifested aspects of concrete operational reasoning, since Lucas and Sierra felt a need to trace the code with specific values − trace it twice, in fact.

It might be argued that Lucas and Sierra traced the code because they were more cautious than the four concrete operational students. Based on our observations across multiple think aloud sessions, Lucas and Sierra are not more cautious than the others, but it is not necessary to rely on such an argument − the transcript demonstrates that their primary way of thinking about code is in terms of specific values within variables. We elaborate upon this point in the remainder of this section.

Before commencing their first trace, Lucas and Sierra have an hypothesis, "*changing the order of the ascending and the descending*?" which is rearticulated as the first trace proceeds to "*It's changing it from ascending order to descending order*". Note, however, that their hypothesis over-specifies what the code does, as the initial values could be in any order. As their first trace proceeds, the hypothesis they articulate describes the code in terms of the specific values they are tracing. It is not Lucas and Sierra's reliance on an explicit, written trace that indicates they are less sophisticated than the four concrete operational students. Rather, it is their focus on reasoning via specific values in the variables. Their approach is inductive, not deductive.

The inductive nature of their thinking is most evident in their decision to confirm their hypothesis by performing a second trace, rather than by simply reading the code. Their choice of initial values for the second trace demonstrates some abstract thinking, as these values will not be altered if their hypothesis is true. Never the less, they felt a need to trace the code with specific values to confirm their hypothesis, rather than confirming the hypothesis by simply reading the code. Furthermore, when selecting the words for their answer, Lucas argued against using the word "change", because "*if it is already in that order, then it will … not change it*". While his argument is correct, Lucas was focusing upon their second trace, where the initial values of the variables remained unchanged. He was not making a general argument in terms of all possible initial values.

## 6 Conclusion

The purpose of this small qualitative think aloud study was to see if we observed problems in novices other than poor English reading/writing skills, misunderstanding the nature of the answer required, and misconceptions about how programs work. Three of our seven subjects (Donald, Lucas and Sierra) did manifest other problems. In neo-Piagetian terms, these three subjects manifested reasoning at the sensorimotor and preoperational stages. Our results are the first direct observational data that is described explicitly in neo-Piagetian terms.

Are Donald, Lucas and Sierra unusually poor students? The grades achieved by these students as shown in Table 1 indicates otherwise. Also, the results

from the earlier quantitative study by Corney, Teague and Lister (2011), and the replication by Murphy, McCauley, and Fitzgerald (2012), suggest that Donald, Lucas and Sierra are not unusual. Furthermore, a multinational study (Lister et al., 2004) established that many students have poor tracing skills at the end of their first semester.

Computing educators need to be more aware of students like Donald, Lucas and Sierra, and should explicitly foster the development of tracing and code comprehension skills in those students. At the very least, our study suggests that when a computing educator encounters a student who struggles to write code, the educator should first check whether that student has adequate tracing and code comprehension skills, before assuming the student is weak at problem solving.

According to constructivist theory, people build new knowledge on the foundation formed by their prior knowledge. We believe that the ability to trace code is the foundation on which abstract reasoning about code is built. That is, while the ability to trace code requires little need to form abstractions of the code, we believe that a novice will not begin to construct correct abstractions in their mind in the absence of the foundational skill of tracing code.

Tracing code is an error prone activity, even for experienced programmers, so the essential skill is not the ability to always get traces exactly right. Instead, we believe the necessary foundation is an efficient strategy for tracing that usually provides a correct answer, perhaps with greater than 50% accuracy, as suggested by Philpott, Robbins and Whalley (2007). While Donald did trace the code correctly at his second attempt, he is an example of a novice who lacks an efficient and well organised strategy for tracing. He did not arrive at a correct explanation until the interviewer intervened to provide initial values for a trace that falsified his initial explanation. Without an efficient tracing strategy, Donald is reluctant to perform more than a single trace, and thus he will struggle to build correct abstractions.

An inability to trace code might explain the bimodal grade distribution that many who teach programming claim to observe (Robins, 2010). Students who have not mastered an effective strategy for tracing code may lack the ability to construct in their mind the abstractions necessary for writing code, and thus can only flounder around, attempting to write code by randomly permuting their code, running it, then repeating that process many times. Those students may form the lower of the two modes in the purported bimodal distribution. Students who have mastered an effective strategy for tracing code have the potential to construct in their minds the necessary abstractions for writing code, and those students might form the upper purported mode.

## Acknowledgements

We thank our student volunteers, Becki, Donald, John, Lucas, Mel, Sierra and Stapler. Support for this research was provided by the Office for Learning and Teaching, of the Australian Government Department of Industry, Innovation, Science, Research and Tertiary Education. The views expressed in this publication do not necessarily reflect the views of the Office for Learning and Teaching or the Australian Government.

## References

Corney, M., Lister, R., and Teague, D. (2011): *Early Relational Reasoning and the Novice Programmer: Swapping as the "Hello World" of Relational Reasoning*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia. pp. 95–104. http://crpit.com/confpapers/CRPITV114Corney.pdf

Corney, M., Teague, D., Ahadi, A. and Lister, R. (2012): *Some Empirical Results for Neo-Piagetian Reasoning in Novice Programmers and the Relationship to Code Explanation Questions*. Fourteenth Australasian Computing Education Conference (ACE 2012), Melbourne, Australia. pp. 77–86. http://crpit.com/confpapers/CRPITV123Corney.pdf

Du Boulay, B. (1989): Some difficulties of learning to program. In Soloway and Sphorer (eds), *Studying the Novice Programmer*. Lawrence Erlbaum. pp. 283–300.

Ericsson, K. A., and Simon, H. A. (1993): *Protocol Analysis: Verbal Reports as Data*. Cambridge, MA: Massachusetts Institute of Technology

Kolikant, Y. and Mussai, M. (2008): *So my program doesn't run! Definitions, origins, and practical expressions of students' (mis)conceptions of correctness*. Computer Science Education, **18**(2): 135–151.

Lister R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. (2004): *A Multi-National Study of Reading and Tracing Skills in Novice Programmers*. SIGCSE Bulletin **36**, 4 (June), pp. 119–150. http://doi.acm.org/10.1145/1041624.1041673

Lister, R. (2011): *Concrete and Other Neo-Piagetian Forms of Reasoning in the Novice Programmer*. Thirteenth Australasian Computing Education Conference (ACE 2011), Perth, Australia. pp. 9–18. http:// crpit.com/confpapers/CRPITV114Lister.pdf

LiveScribe (2011): Retrieved October 24, 2012, from http://www.smartpen.com.au/

Murphy, L., McCauley, R. and Fitzgerald, S. (2012): *Explain in Plain English' Questions: Implications for Teaching*. SIGCSE'12, Feb 29–Mar 3, Rayleigh, NC, USA.

Philpott, A, Robbins, P., and Whalley, J. (2007): *Accessing the Steps on the Road to Relational Thinking*. 20th Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ'07), Port Nelson, New Zealand. p. 286.

Robins, A. (2010): Learning edge momentum: a new account of outcomes in CS1. *Computer Science Education*, **20**: 1, pp. 37 – 71.

Simon and Snowdon, S. (2011): *Explaining Program Code: Giving Students the Answer Helps – But Only Just*. Seventh International Computing Education Research Workshop (ICER 2011), Providence, Rhode Island, pp. 93–99.